LANGLEY
GRANT
IN-61-CR
110419
11P

DEPARTMENT OF COMPUTER SCIENCE
COLLEGE OF SCIENCES
OLD DOMINION UNIVERSITY
NORFOLK, VIRGINIA 23508

SOFTWARE RELIABILITY PERSPECTIVES

By

Larry Wilson, Principal Investigator

and

Wenhui Shen, Co-Principal Investigator

Final Report
For the period ending March 31, 1988

Prepared for the
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665

Under
Research Grant NAG1-750
Gerard E. Migneault, Technical Monitor
ISD-Systems Validation Methods Branch

Submitted by the
Old Dominion University Research Foundation
P. O. Box 6369
Norfolk, Virginia 23508

December 1987

(NASA-CR-181523) SOFTWARE RELIABILITY PERSPECTIVES Final Report, period ending 31 Mar. 1988 (Old Dominion Univ.) 11 p CSCL 09B

N88-12952

Unclas
G3/61 0110419

Software Reliability Perspectives

Larry Wilson
Wenhui Shen
Department of Computer Science
Old Dominion University
Norfolk, VA 23508-8508

Abstract

Software which is used in life critical functions must be known to be highly reliable before installation. This requires a strong testing program to estimate the reliability, since neither formal methods, software engineering nor fault tolerant methods can guarantee perfection. Prior to the final testing software goes through a debugging period and many models have been developed to try to estimate reliability from the debugging data. However the existing models are poorly validated and often give poor performance. This paper emphasizes the fact that part of their failures can be attributed to the random nature of the debugging data given to these models as input, and it poses the problem of correcting this defect as an area for future research.

Additional Key Words and Phrases: Growth Models, Software Reliability, Debugging Graph

0. Introduction

Software reliability has become a major concern in the field of computer science. Attempts have been made to construct more reliable software by improved software engineering techniques, formal methods, and/or fault tolerant methods. Also, models have been developed which will predict the failure rate software will exhibit in future performance. We will present a survey of the previous research on software reliability and try to place it in perspective. A major problem with the reliability models will be identified and new directions for future research will be suggested.

This paper has nine sections. Sections one and two exhibit the reason we need be concerned with software and terminology necessary to address the problem. Section three describes existing data sets of value and is followed by sections on models, formal methods, software engineering, and formal methods, together these five sections describe what has been done to date. Section eight describes a common problem for all the models and is followed by the section on conclusions.

1. Motivation

Complex applications of computers which are life critical demand a high degree of reliability from their computer systems. This problem should be viewed as having two parts, namely the development of highly reliable systems and the assessment of that reliability. In order to implement life critical

This research was in part supported by NASA Grant 1-750.

systems we require a certain degree of confidence in their expected performance. It would of course be acceptable to be able to say a system is perfect, that it will never fail to perform correctly, but we do not expect that of complicated systems and hence we must ask how reliable are they. Further, it is not sufficient to say very reliable, since this will not allow us to choose the better of two very reliable systems or even to say if either is worthy of life dependency. Rather, we must adopt some scheme of quantifying reliability.

As a preliminary approach, it is desirable to split reliability assessment into separate but not equal problems of hardware reliability and software reliability assessment. The hardware problem is well understood and correctly modeled as a Poisson Process, with constant arrival rate, during the useful life of the machine (after burn in and prior to old age). This is modeled by the known bathtub curve and is well supported by experiments and experience.

Software reliability, the other component of system reliability is more difficult and less well understood. Early attempts to model it have tried to force the characteristics of hardware onto software without justification. The terminology itself is too suggestive of hardware, since it suggests more similarities than exist. This is unfortunate but we should be aware that the term 'software reliability' is not the same as the traditional term 'reliability'. Of course, when previous results fit, we should exploit that fact, but we must not force old attributes onto a new problem area, simply because we know how to solve the old problem. The scientific method is to approach a new problem area by observation, experimentation, data collection, and analysis. Many opinions have been formed from observation and much analysis has been done but few scientific experiments have been performed and thus we have very limited data sources with which to validate our models. We view the valid existing data as preliminary, hence we will refer to observed trends from the data rather than theories or facts.

2. Terminology

A. Software Development Phases

For our purposes we describe the modern development process as consisting of five, not necessarily distinct, phases. We label these phases as specification, programming, debugging, testing and implementation and briefly describe each phase below.

Phase one consists of writing the specifications. This phase uses critical knowledge about the intended application and the role to be played by the computer system in order to produce a formal descriptive document of the task to be performed by the computer.

The second phase, programming, consists of translating the specifications into software for the targeted hardware. It includes design, module building, testing and debugging, program integration, and integration testing.

Debugging is an ongoing process throughout phase one and two, however, we would like to consider that phase three, debugging, begins after all of the code is written and integrated, but is still under the control of the programmer. The goal during this phase is to find and eliminate errors in the system.

Validation, phase four, consists of acceptance testing and is used to certify or attempt to certify that the system is reliable enough to use. This may be done with any combination of laboratory testing against the specifications, laboratory testing against a simulation, or of implementation testing. Unfortunately implementation testing, which checks the specifications in a real environment, is very expensive. Simulation tests model an implementation and then test the software against the model, hoping to get the benefits of implementation testing more cheaply but they can never be equivalent to implementation testing and each simulation must itself be validated in order to be useful. Laboratory tests versus the specifications are relative inexpensive but also less satisfying than implementation or simulation tests.

Phase five, implementation, begins after successful validation and lasts until the software is abandoned or modified. It should be understood that modified software is a new product, and must be evaluated as to reliability and hence modification logically dictates a return to phase four or even to phase three. No change should be viewed as safe without testing, since one cannot presume to guess the ripple effects of a change through the various layers of software utilized in creating source code.

B. Reliability Assessment

Software reliability has been given many definition including one which defines it as the probability that a given piece of software will perform correctly in the environment for which it was intended for a specified period of time, one which defines it as mean time to failure and one which defines it as the number of failures expected over some period of time. Since variations in the input frequency, input usage, or other parameters can effect performance whether it is measured in cpu cycles or calendar time, we recommend that software be modeled as a mathematical function, which for given input is expected to produce a desired result. Thus, reliability is the probability that for the next input, the system will perform correctly. It is assumed that the software to be measured has a set of specifications which determine the input space including the usage distribution, the correct output for each legal input, what to do with illegal inputs, and sets time constraints for these computations.

Definition:

The Fundamental Software Reliability (FSR) measure is the probability that the given piece of software will give the correct output for the next randomly chosen input according to the input usage distribution Q. Where Q associates with each potential input value, the probability of that input being chosen.

This definition forms the basis for the Nelson model [20]. It is simple to understand and easy to approximate. Approximations may be made by drawing random samples from the input space as dictated by the usage distribution and testing them.

The usage distribution Q must be derived from the planned implementation and therefore must be presumed to vary from one application to the next. Without loss of generality, we choose to view the input distribution Q as being uniform for the rest of this paper. To justify this, consider that for any problem with non-uniform Q, we obtain an equivalent problem in which the input space is made of copies of the original inputs in proportions dictated by Q. Thus, if we calculate the FSR for the new software applied to the new problem with uniform distribution of input we get the correct value for the original problem with an arbitrary distribution.

For software which requires memory from one input cycle to the next it would be more natural to test by taking a random walk through the input space rather than selecting isolated random inputs. However, we can also get results by treating the memory as part of the next random input in order to simulate memory.

For some applications, the specifications and/or usage distributions may be functions of time or of previous inputs. For time dependency we would allow FSR to be a function of time or an average over time.

In mathematical terms, we are viewing a piece of software as a function F which for any x in the domain (input) calculates

$$y = F(x),$$

where y is in its range (output), subject to time constraints. The performance of F is to be measured against a set of specifications, S, which determines a function from its domain Ds to range Rs as well as specifying time constraints. Thus for any x in Ds, we say our software fails if x is not in the domain of F or if F(x) fails to equal S(x) (x is considered to not be in the domain of F, if the time constraints are not met). Precise calculation of FSR would require the validation of each point in Ds, which will normally force us to approximate FSR by random sampling from Ds since the input space may be enormous.

3. Experiments and Data

The development of software reliability models has been hampered by inadequate data sets which are often imprecise and subject to influences other than software. Researchers have often used data from either a normal debugging operation or from operational data. In each of these cases the data is influenced by the particular input stream encountered and accurate recording of the data is not the primary goal of the participants. Further this data is not reproducible under normal circumstances. These facts infer that one should not to make life critical decisions based on models which are validated only with this type of data.

Phyllis Nagel [18,19] has developed a method for producing time-tagged data in a reproducible, repetitive run experiment. The technique consists of using random inputs to discover 'n' bugs together with a fix for each bug found. Next the fixes are removed and the program is debugged again using random inputs, but exploiting the fixes already known. Each debugging results in the removal of most of the 'n' known bugs and is called a repetition. After fifty repetitions, we have a firm grip on the failure rate of most of the known bugs. Dunham [5] has conducted similar experiments using modifications and extensions of the Nagel experiment. These few data sets are also relatively small when considered against the problem of trying to understand software reliability, but they do remove the randomness from the debugging process and they are reproducible in a statistical sense.

These data sets are too small and limited in nature to make positive assessments about software reliability, but the appear to form counterexamples to models which assume all bugs have equal failure rates. In fact when the data is used in these models, they regularly predict that the last bug has just been found. That is if we know of ten bugs, but use the data from the first five, then the model predicts five total bugs, but by adding the data from the sixth bug, we can get the model to predict six total.

Knight and Leveson [10] have conducted what appears to be an excellent experiment, although for reasons of their own they have refused to release the data. Their experiment produced 27 independently developed versions of code written to the same specifications. Their analysis concluded that the independently developed programs did not fail independently, and thus for fault tolerant software architectures it must not be assumed that components will fail independently when computing reliability. Eckhardt and Lee [7] draw the same conclusion from a theoretical view point.

One trend, which has been observed in the quality data sets which exist, has been labeled as the log linear trend. That is if each bug in a program were associated with a mean time to failure for that particular bug and the bugs were arranged in order of decreasing failure rates, if we plot the rank of the bug in this ordering versus the logarithm of its mean time to failure, then the result appears to approximate a straight line. We use the terms trend and appears since we recognize that these samples are too limited to do more than suggest possibilities. However, this log linear trend has been observed in the Nagel work, the Dunham work and the Knight work. Further, Paul Ammon [1] working with John Knight has observed a log linear trend in the failure rate of seeded bugs. Thus there is some hope that this trend will turn out to be a property of software in general.

The Dunham [5] experiment led to the discovery of bug interaction. By bug interaction we mean the phenomenon in which the set of error causing inputs for each of two or more bugs is different when the other bugs are present than it is when they are not. Bug interaction raises the possibility that an otherwise perfect program might be more reliable with two bugs in it than if we were to fix just one of them. That is, a fix which does not insert a new error into the code may cause the code to be less reliable if it were used. Thus software reliability growth models must account for the possibility of negative growth. The existing data sets indicate that this phenomenon is not rare in real programs and therefore it should be accounted for in reliability estimations.

4. Models

A. Validation Phase

The purpose of the validation phase is to try to predict the reliability which the software will exhibit during the operational phase. The only sound theoretical model [21] for doing this is the Nelson model [20] which is based on random testing governed by the the input distribution. This model is criticized for requiring too many test cases, failing to take into account the continuity of the input domain, failing to exploit special test strategies, and failing to exploit complexity measures of the software to determine length of testing needed. We disagree with the last three objections based on recent research and observations some of which is not yet published.

Continuity, is in general an unwarranted assumption, we have recently observed [14] a failure producing pattern in an input space which looked like an irregular checker board, with the dark regions being the failure producing inputs. By refining the grid on the input space and examining a sub region blown up, we again observe the irregular checker board pattern, that is neighboring points cannot be used to predict whether a specific input will cause failure or not.

Special test patterns belong in the debugging phase if they belong at all. During debugging we wish to discover as many of the problems in the software as possible and of course repair them. This entire process should be completed with a minimum of time and expense, hence we may want to use special test patterns here in order to increase the reliability of the product which is delivered to be validated. If we use the same special test patterns in validation then we may get a false reading on reliability since those are precisely the input patterns which have already been heavily tested.

Complexity measures of software and their relationship to reliability is a controversial subject in the field of software engineering. Most of the metrics tend to predict nicely the length of the code and none have been scientifically calibrated to the reliability of software.

For reliability estimation purposes we believe that their is no known shortcut during validation. If one wants to have confidence in the reliability predicted using data generated by testing then one must pay for tests similar to those required by the Nelson model.

B. Debugging Phase

Reliability growth models attempt to predict operational reliability from data gathered during the debugging. This prediction may be used to determine the end of the debugging phase and/or to predict operational reliability without further testing. We will selectively discuss models from a 1984 paper by Ramamoorthy and Bastani [21] as well as the Musa-Okumoto model [17] in terms of their promise in light of recent experiments. Readers interested in the more complete survey and or the implementation of these models are referred to the Ramamoorthy paper.

The General Poisson Model [2], which generalizes the Jelinski-Moranda linear de-eutrophication, the Shooman, and the Schick Wolverton model, is based on the assumption that all errors contribute equally to the failure rate of the program. This is intuitively wrong and data from experiments [5,18,19] also shows that it is wrong. Further there are more promising models, which are descendants of the General Poisson models and are available today.

The Littlewood model [11] is criticized in [21] for assuming additivity and independence of failure rates associated with individual faults. These properties are in general not compatible.

The Moranda Geometric De-eutrophication model [16], the Input Domain model [21] and the Musa-Okumoto model [17] all exhibit a log linear relationship when the sequence of expected inter-failure times from the debugging process is plotted versus the sequence index. This pattern has a certain intuitive appeal since the errors with higher failure rates are expected to be found first and the pattern has been observed in data [5,18,19] as well as in seeded errors [1].

No growth model has demonstrated that it can be used with a high degree of confidence to predict operational reliability from data generated in the debugging phase in a general setting. Further, if we possessed a model which made this prediction accurately we would still be unable to dispense with validation testing since the debugging is under control of the developers of the code and they will always have strong incentives to show the code is acceptable rather than unacceptable. If we get an independent group to debug the code then we suffer because they are ill equipped to make the necessary changes and they too are motivated to find the code has achieved the desired reliability. Hence, the most use we foresee of growth models is to help developers decide when to submit their product to the validation group which should be independent of the developers.

5. Formal Methods

Formal methods, including correctness proving, are not prepared to deliver perfection for medium to large programs at this time and there is reason to doubt that they ever will. The technology does not exist for automated proving of software at this time and if it did, we would be obligated to query the reliability of the theorem prover itself. One famous effort at establishing the validity of correctness proving was the multi-year, large dollar effort by SRI on the SIFT operating system. SRI claimed many things but did not prove the operating system correct nor did they deliver a functioning operating system, yet this is often quoted as a success of theorem proving. The value of formal methods has not been exhibited for realistic systems, using either automatic provers or manual. Yelowitz and Gerhardt [8] cast doubts about relying on proofs for perfection in simple applications much less in complicated ones. We see some possibility for saving time and money in the debugging process if formal methods are used in the development, but that gain is not guaranteed nor quantified and there will be extra expense in developing software in this manner, since most software developers are not sophisticated theorem provers nor do they think in those terms. Further, it has yet to be assessed just what the effect of proving will have on the testing phase, unless it is perfect then the reliability must be assessed and no one should expect it to be perfect for medium to large systems handling complicated problems.

Formal methods, with or without theorem provers, may produce code which originally has fewer bugs, but perfection is unlikely and testing will be necessary. Suppose we want software that fails at most one time in a million inputs, that normal coding methods produce code such that at the start of debugging it fails once in one hundred inputs, and that formal methods produce code which fails only once in ten thousand inputs. If after one week of debugging the normally produced code is failing at the rate of one in ten thousand, then the net gain in time and money by using formal methods would be one week of debugging time minus whatever extra costs would be incurred in time and money by using formal methods. This fictitious example describes some of the items we should consider before adopting formal methods. Of course, if the formal methods produce code guaranteed to satisfy the reliability criteria, without debugging or testing, then we might have a significant savings. Many proponents of formal methods, point to SIFT as an example of what can be done, having observed SIFT being installed at NASA, where it took well over a year of debugging and testing to get it operational and its reliability is still unknown, we are certain that formal methods should not completely displace testing in the near future.

6. Software Engineering

Some progress has been made by improving techniques for developing [3] large software systems, but here has been no significant breakthrough in the area of reliability assessment. That is, improved specification design techniques and test case selection algorithms have increased productivity and made it cheaper to identify and remove some of the bugs in a possibly smaller set of bugs than would have been produced without these techniques. However, specifications are still likely to be ambiguous and imprecise. It also seems probable that most of the bugs prevented in the design stage would have been removed early during debugging stage anyway, since these improvements have come about because of previously observed errors, known to occur often in large software projects and there are test strategies which specifically look for common errors.

The increased modularity plus component development and testing as well as integrated testing as the system is assembled, makes it cheaper to produce and maintain highly reliable software and it will prevent many bugs from reaching the debug phase. It is far less expensive in time and money to debug components than removing the same bugs from the final software. Also those bugs which are found during debugging will be more quickly analyzed and fixed because of increased modularity, increased documentation, and better understanding of the system by the builders through their experience in working with the components and assembling process.

Specialized testing techniques, such as boundary testing, stratified testing and path testing will help discover remaining bugs more quickly than would random testing. However, after these results are in we still must assess reliability and since no predictor exists to relate any special test strategy to reality as closely as random testing is expected to be related, we still must resort to random testing to assess reliability. Worse case testing would be acceptable but we do not know how to identify the worse case in general and the situation may have been obscured by the debugging strategy employed. We could use random input for the debugging phase but that invokes the likelihood of more bugs in the product being assessed for reliability.

7. Fault Tolerance

Recovery block schemes and n-version programming schemes are used to develop fault tolerant software. Both schemes employ multiple versions of the software but they differ in the way they select an answer from the versions and they differ in the way they are normally configured for implementation. The n-version architecture normally runs n-versions in parallel and a voter selects the proper output, while in recovery block schemes it is normal to run a single version of the code, test it against the error detector, and only if it fails the test does a second version of the code run and get tested. Either scheme may be nested within itself, although it is more common to do so with recovery blocks. For more information see Wild [24].

Each of the above schemes for tolerating faults promises great gains in reliability if we believe that the multiple versions fail independently. However, the data in Knight-Leveson [10] disputes this possibility and theory developed in Eckhardt-Lee [7] also contradicts the practice of assuming independent failures amongst independently developed software. It seems more likely that there are easy cases and hard cases for the software to handle and that in multiple versions, all versions will have trouble with the hard cases while getting the easy ones correct, thus we expect dependency amongst the failures.

Littlewood-Miller [13] shows a new theoretical model for planned diversity of versions, but can only prove that this might be better for systems of n-programs where only one needs to be correct. They show that with diversity a one out of n system may perform better than would be predicted by assuming independent failures, they do not show how to exploit this in a realistic application, but one can visualize using this property for systems where it is critical never to be wrong and you have the option of not doing anything. They also show that if two out of three are needed to be correct then diversity will possibly not help and the system is not expected to perform as if the versions failed independently.

If one could build perfect fault tolerant systems then that would solve the reliability problem of software. However, it is unlikely that we can mass produce such systems, so we must assess their reliability, just like any other software system except they will tend to be larger and more complex due to fault tolerant features. We can try and assess the reliability of the component versions and to use these numbers to calculate the reliability of the fault tolerant system. To do this we need the reliability of each of the n-versions and of the acceptance test plus a scheme for using these component reliabilities to predict the system reliability. Fault tolerance does not necessarily solve the reliability problem, it merely modifies it and possibly complicates it. To say it another way fault tolerance may or may not make more reliable systems but it does not obviate the need for reliability assessment.

8. The Debugging Graph

Let P be a program which at the start of the debugging process would be perfect if each of N fixes f_1, f_2 , ..., f_N were applies to N bugs b_1, b_2, ..., b_N respectively. We indicate alternate versions of the program by adding subscripts to P to indicate which fixes have been installed, thus P_{abc} is the version of P which results when precisely fixes f_a, f_b, and f_c have been installed. We obtain a directed graph by letting the versions be nodes and letting each edge represent the installation of one new fix.

This results in a graph with initial node P at level N and terminal node $P_{1,2,\ldots,N}$ at level 0. Thus the level of a node in the graph is the same as the number of bugs remaining in that node. If we debug by random testing until we have found n bugs and their corresponding n fixes, then without loss of generality we will assume that f_1, f_2, ..., f_n have been installed in that order (we could always renumber them). If multiple fixes are required for a single failure then we arbitrarily choose the order in which they are applied. Thus after possibly renumbering some nodes in G we obtain a subgraph G' which begins at P and terminates at $P_{1,2,\ldots,n}$.

We would like to be able to estimate the unknown features of G based on the features of G'. In particular we would like to know the reliability of $P_{1,2,\ldots,n}$ and/or how much more time we expect to spend in the debugging process in order to reach a desired level of reliability.

The information available to us when we arrive at P is typically the MTTF data for versions P, P_1, ..., $P_{1,2,\ldots,n}$. Since existing data set indicate an exponential decline in the failure rate of individual bugs arranged in the order of decreasing MTTF, we would like to use this data to predict the MTTF of $P_{1,2,\ldots,n}$. Unfortunately, the debugging data is subject to random variations, that is is we were to debug again by making a new selection of random inputs we would not expect to get the same data as we originally obtained. If we make predictions based on debugging data from one pass at removing the bugs then we will somehow have to account for all of the uncertainty in the data due to the random choices of input.

The process of removing n particular bugs can be characterized as taking a random walk on G' from P to $P_{1,2,\ldots,n}$, with the walk governed by the probabilities of choosing each individual arc. For example if b_1 has twice as large a failure rate in P as b_2 then we are twice as likely to go from P to P_1 as we are to go to P_2. Even though there is a most likely path from P to $P_{1,2,\ldots,n}$, it is not unusual to choose a different path when debugging and end up trying to make predictions based on this secondary path. The variance in path selection must be accounted for by the reliability models and it is our conjecture that this variance possibly plays a large role in the poor performance of the models.

9. Summary

The randomness present in data generated by the debugging process has been identified as a problem for software reliability models. Future research is needed to understand the extent of the damage due to randomness and to propose and evaluate methods of removing the randomness from the data.

8. References

1. Paul Ammon and John Knight, "An Experimental Evaluation of Simple Methods for Seeding Program Errors," Computer Science Report, Tr-85-08, Univ, of Va, Charlottesville, Va. July 1, 1986.

2. J. E. Angus, R. E. Schafer, and A Sukert, "Software Reliability Model Validation," in Proc. Annu. Rel. and Maintainability Symp., San Francisco, CA, Jan. 1980, pp. 191-199.

3. Fred Brooks, "No Silver Bullet- Essence and Accidents of Software Engineering," Information Processing 86, H.-J. Kugler(ed), Elsevier Science Publishers B. V. (North Holland), IFIP, 1986.

4. T. A. Budd, A. DeMillo, R. J. Lipton, and F. G. Sayward, "Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Programs,," Proceedings ACM Symposium on Principles of Programming Languages, 1980.

5. Janet R Dunham, "Experiments in Software Reliability: Life- Critical Applications," IEEE Transactions on Software Reliability, Vol SE-12, No 1, Jan 1986, pp110-123.

6. Janet R Dunham and John L. Pierce, "An Experiment in Software Reliability," NASA Contractor Report 172553, March 1985.

7. Dave E. Eckhardt and Larry D Lee, "A Theoretical Basis for the Analysis of Multiversion Software Subject to Coincident Errors," IEEE Transactions on Software Engineering, vol., SE-11, Dec 1985.

8. S. L. Gerhart and L. Yelowitz, "Observations of Fallibility in Modern Programming Methodologies," IEEE Transactions on Software Engineering, vol. SE-2, No. 3, Sept., 1976.

9. Z. Jelinski and P. Moranda, "Software Reliability Research," in Statistical Computer Performance Evaluation, W. Freiberger, Ed. New York: Academic, 1972, pp. 465-484.

10. John C. Knight and Nancy G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multiversion Programming", in IEEE Transactions on Software Engineering, vol SE-12, No 1, Jan 1986.

11. Bev Littlewood, "A Bayesian Differential Debugging Model for Software Reliability," Dep. Math City Univ., London, England, June 1979; also in Proc. COMPSAC 1980, Chicago, Il, pp. 511-519.

12. Bev Littlewood, "How to Measure Software Reliability and How not to ....," IEEE Trans. Rel., vol R-28 pp 103-110.

13. Bev Littlewood and Douglas R. Miller, "A Conceptual Model of Multi- Version Software", Proceedings of 3rd IFTCS, Bremerhaven, Sept., 1987.

14. G. E. Migneault and B. Becher, unpublished research which is ongoing at NASA, Langley Research Center, Hampton, Va.

15. Douglas R. Miller, "Exponential Order Statistic Models of Software Reliability Growth," in IEEE Transactions on Software Engineering, vol SE-12, No 1, Jan 1986, pp12-24.

16. P. B. Moranda, " Prediction of Software Reliability During Debugging," Proceedings of the 1975 Annual Reliability and Maintainability Symposium.

17. J. D. Musa and K Okumoto, "A Logarithmic Poisson Execution Time Model for Software Reliability Measurement", 1984, IEEE.

18. Phyllis M. Nagel and James A. Skrivan, "Software Reliability: Repetitive Run Experimentation and Modeling" NASA Contractor Report 165836, Feb 1982.

19. Phyllis M. Nagel, Fritz W. Scholz and James A. Skrivan, "Software Reliability: Additional Investigations Into Modeling with Replicated Experiments," NASA Contractor Report 172378, June 1984.

20. E. Nelson, "Estimating Software Reliability from Test Data," in Microelectronics and Reliability, vol. 17. New York: Pergamon, 1978, pp. 67-74.

21. C. V. Ramamoorthy and Farokh B. Bastani, "Software Reliability--- Status and Perspectives," in IEEE Transactions on Software Engineering, vol. SE-8No. 4, July 1982, pp. 354-371.

22. G. J. Schick and R. W. Wolverton, "Achieving Reliability in Large Scale Software Systems," in Proc. 1974 Rel. and Maintainability Symp., Los Angelos, CA, Jan 1974, pp. 302-319.

23. M. L. Shooman, "Probabilistic Models for Software Reliability Prediction," in Statistical Computer Performance Evaluation, W. Freiberger, Ed. New York: Academic, 1972, pp. 485-502.

24. Christian Wild, "Concepts and Terminology for Analysis of Fault Tolerant Hardware and Software," Interim Report Phase 2, Task TTD#2 of Contract no DTRS-57-84-C-0013, Mandex Inc, Springfield, Va, Oct 1986.